(4)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NW-LIS-89-12-03	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Drawing Wireslisp		5. TYPE OF REPORT & PERIOD COVERED Technical
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Zhanbing Wu, Carl Ebeling		8. CONTRACT OR GRANT NUMBER(s) N00014-88-K-0453
9. PERFORMING ORGANIZATION NAME AND ADDRESS Northwest Laboratory for Integrated Systems, University of Washington, Dept. of Comp. Science, FR-35 Seattle, WA 98195		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS DARPA-ISTO, 1400 Wilson Boulevard, Arlington, VA 22209		12. REPORT DATE December 1989
		13. NUMBER OF PAGES 18
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Office of Naval Research - ONR, Information Systems Program - Code 1513: CAF, 800 North Quincy Street, Arlington, VA 22217		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this report is unlimited.

Cb

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Lisp, TLisp, Graphical description, procedural description, VLSI circuits, schematic, parametrized schematic.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

WireLisp is a hierarchical circuit structure description language that combines the intuition of schematics and the generality of procedural description. It is different from other similar tools in that the schematics and procedural descriptions are closely intertwined. More specifically, WireLisp is embedded in Lisp but provides graphical constructs for the most common procedural constructs. A WireLisp program consists of a set of device definitions, each described in the most convenient way:

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-LF-014-6601

CONTINUED ON BACK...

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

#20 ABSTRACT, (continued from front page)

Lisp expressions may be embedded in schematics and schematics may be embedded in Lisp as well. This allows descriptions to be highly expressive, yet easily specified and understood. This manual describes how to use the interactive graphic editor Xdp to make WireLisp drawings.

Drawing Wirelisp

Zhanbing Wu and Carl Ebeling

Technical Report #89-12-03

Department of Computer Science and Engineering, FR-35
University of Washington, Seattle, WA 98195 USA

Drawing Wirelisp

Zhanbing Wu and Carl Ebeling

Technical Report #89-12-03

Department of Computer Science and Engineering, FR-35
University of Washington, Seattle, WA 98195 USA

Drawing Wirelisp

Zhanbing Wu Carl Ebeling

Contents

1 Introduction

This document describes how to use **Xdp** to prepare Wirelisp drawings. The reader is assumed to have some basic knowledge about **Xdp** and to have read the *Wirelisp Manual* which describes procedural circuit descriptions.

Xdp is an interactive graphic editor for drawing circuit schematics. It supports a variety of general graphic objects including lines, text, pins, circles, arcs, etc.. It also has a macro concept whereby a group of related objects can be bound into a single named object called *symbol*. But **Xdp** has no understanding of the semantics of circuit structures. It is up to **dp2wl**, a drawing analyzer, to extract the Wirelisp procedural representation from the schematics.

dp2wl supports hierarchical circuit design in the same way as Wirelisp. There is a one-to-one correspondence between Wirelisp programs and Wirelisp drawings. Each drawing is mapped into exactly one device definition. That is, a Wirelisp design consisting of 10 devices will be defined in 10 drawings. A drawing may include arbitrary Lisp expressions which are represented as **Xdp** strings. But typically, most of the information is represented graphically, with text mostly used to describe repetitive or conditional structures.

A Wirelisp device definition consists of two major parts: the *device header* which defines the device name and formal parameters, and the *device body* which describes the device implementation. This is directly mapped into a drawing: The top-most symbol represents the device header, and everything else corresponds to the implementation. For example, the following Wirelisp procedure can be drawn as Figure 1:

```
(-device- (foo in1 in2 in3 out)
      (local temp)
      (-I- cnand temp in1 in2)
      (-I- cnor out temp in3))
```

A drawing typically consists of items on two layers: **STANDARD** and **COMMENT**. All items on the **COMMENT** layer are ignored. Normally, the user places descriptive information on this layer, such as the device name, the author name, and a frame. For example, the frame and the big letters in Figure 1 are on the **COMMENT** layer. Since items on this layer look just the same as those on other layers, the user is advised to draw them on the **STANDARD** layer first and then use the command 'n' (changing the parameter **layer**) to move them to the **COMMENT** layer.

Comment text can be placed on the **STANDARD** layer also by preceding each string with one or more semicolons, which is the Lisp comment character. The position of these strings is irrelevant to **dp2wl**.

2 Device Definition

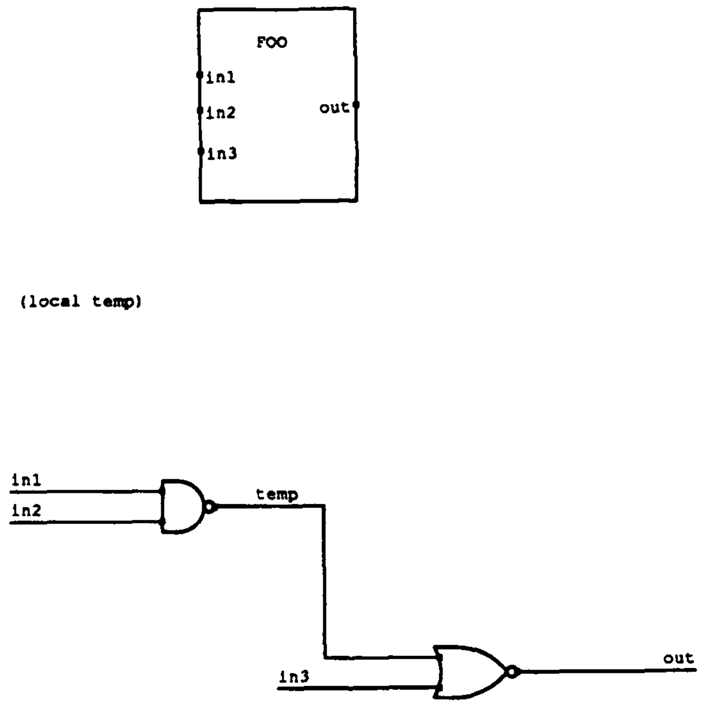
A device is represented by a symbol. The device being defined must be placed at the top of the drawing. Its name is derived from the prefix before the first period of the symbol name. This allows more than one definition for a device (see Section 3.4). For example, the name of the top symbol in Figure 1 is `foo`, which is also the name of the defined device.

2.1 Formal Signal Parameters

The formal signal parameters are represented by pins packed inside the symbol. For example, Figure 1 defines the device `foo` which has 4 signal parameters: `in1`, `in2`, `in3` and `out`. Usually pins are laid around the periphery of the symbol.

The formal signal parameter names naturally derive from the pin names. The basic method of associating a name to a pin is placing the string sufficiently close to the pin. This can be strictly defined as:

Rule 1: string-touch-pin-inside-symbol



UW/LIS

Figure 1: *A Simple Wirelisp Drawing.*

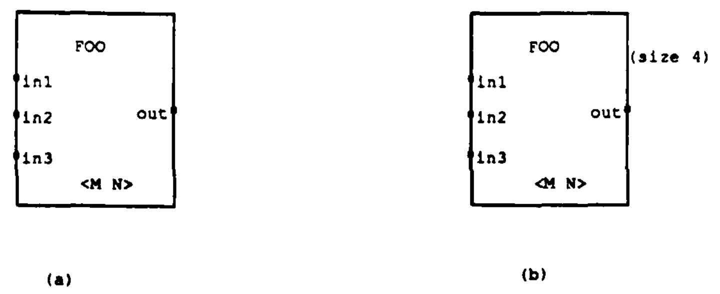


Figure 2: *Examples of the Top Symbol*

1. ***A pin placed on the top (bottom) boundary of the symbol will be associated with the closest string below (above) whose left and right ends span the pin's X location.***
2. ***Otherwise, a pin placed to the left (right) of the symbol center will be associated with the closest string to its right (left) whose top and bottom corners span the pin's Y location.***
3. ***In both cases, the string must be within 12 pixels of the pin.***

The signal parameters are ordered according to increasing pin number. Therefore, it is very important that symbols representing a device are the same everywhere so that the parameter lists agree.

2.2 Formal General Parameters

The formal general parameters are specified by a single string which is a list of space-separated parameter names enclosed by '(' and ')'. The order of the listed names defines the order of the formal general parameters. The string must be packed inside the symbol and should not be too close to any pin. For example, the symbol shown in Figure 2 (a) has two general parameters and will be translated into:

```
    (-device- (foo in1 in2 in3 out M N) ...)
```

2.3 Optional Parameters

An optional parameter is defined by declaring both a name and default value. It is represented by a string in the form:

(*name value*)

The user should place the string sufficiently close to (or over) the bounding box of the symbol. This can be strictly defined as:

Rule 2: string-touch-box

Either the lower left corner or the lower right corner of the string[1] *should be inside the area enclosed by the box.*

For example, the symbol shown in Figure 2 (b) has one optional parameter named **size** which has the default value **4**.

[1] These corners become visible when the string is selected.

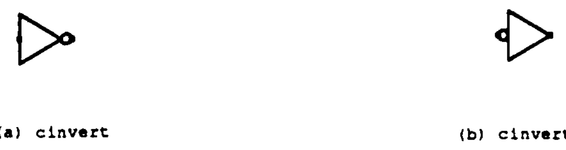


Figure 3: *Icons for* **cinvert**

2.4 Multiple Definitions

It is often convenient to have several different representations for the same device. For example, an inverter has two different symbolic representations as shown in Figure 3. This is done by defining two symbols whose names differ only after the first period, for example, **cinvert** and **cinvert.i**. Only one of these symbols may be placed on **STANDARD** layer. The other symbols must be put on the **COMMENT** layer. (When they are copied to another drawing, they will end up on the current layer, which is typically the **STANDARD** layer.)

3 Device Implementation

The implementation part of a drawing is basically an enumeration of symbols representing the constituent devices, with their connection points wired in the obvious way.

3.1 Device Instantiation

A device instantiation is represented by a symbol designating its definition together with the actual parameters:

- An actual signal parameter is normally specified by connecting the pin to the signal wire. This will be explained in detail.
- All the actual general parameters must be specified in a single string separated by spaces. The string must be enclosed by '(' and ')' and placed sufficiently close to the symbol (see Rule 2).
- Actual optional parameters can be specified in the same way as for the top symbol: A string with the name-value pair of the optional parameter is placed over the symbol (see Rule 2).

The actual signal parameters are ordered by increasing pin numbers. The order of actual general parameters is the same as given by the user. Again, it is very important that the symbol definition used is up-to-date. Use '**y**' command to make sure.

3.1.1 Wires

A wire is drawn as a line of width 1. It can be of any length and slope. Wires are connected through their end points. This can be strictly defined as:

Rule 3: line-touch-line
Either end of one line should be within 3 pixels of some point on the other line.

Figure 4 (a) shows several examples of wire touching. Wire touching is transitive, that is, touching wires form an equivalence class.

One consequence of this rule is that lines that *cross* (no endpoint touching) are not connected (Figure 4 (b)). If crossing lines are meant to touch, then one of the lines must be split in two. Moreover, the two lines should be drawn so that it is obvious that it is not a single crossing wire. We suggest that to make two crossing lines touch, draw a short diagonal line whose endpoints lies on each of the two lines. (See Figure 4 (c))

A wire is connected to a pin if it is sufficiently close to the pin. This can be strictly defined as:

Rule 4: line-touch-pin
The pin should be within 3 pixels of some point on the line.

The **Xdp** gravity feature should help the user to make such connections. It is assumed that there is at most one wire connected to a pin. Otherwise, **dp2wl** will choose the first wire it sees to connect the pin and warn the user about other wires.

A name is associated with a wire by placing it sufficiently close to the wire. This can be strictly defined as:

Rule 5: string-touch-line
Either the lower left corner or the lower right corner of the string should be within 3 pixels of some point on the line.

It is assumed that all strings are horizontal. For most common cases, this rule means that touching strings *sit on* horizontal wires, or lie just to the left or right of vertical wires. Figure 5 gives some examples. It is ambiguous if a string can be associated with more than one wire. In that case, **dp2wl** will assign the name to only one wire and warn the user.

Wires that connect are considered to carry the same signal. If the user places more than one name on a wire, **dp2wl** will choose a name for the wire and then create aliasing (`-connect-`) for the remaining signals.

3.1.2 Automatic Signals

The user can avoid declaring every signal by using unnamed wires or annotating wires with special symbols as shown below.

If an unnamed wire is found, **dp2wl** automatically creates a unique name of the form

    $⟨device-name⟩⟨num⟩

where, *device-name* is the name of the device being defined, and *num* is a unique integer. This name is given to the wire and declared as a *short simple* signal.

To specify a cable of size N, the user may annotate the wire with a string of the form

```
/N
```

where N may be any integer expression. **dp2wl** will create a unique name for the wire and declare the signal as a cable with the structure

```
(BUS 1 N)
```

Furthermore, if the name contains BUS indices, such as

```
/N[i]
```

dp2wl declares a signal as for the `/N` case but names the wire by appending the index which can be any integer expression. Typically, `i` is the control variable of a loop structure. Therefore, the `i`'th component of the cable will be referenced during the `i`'th iteration.

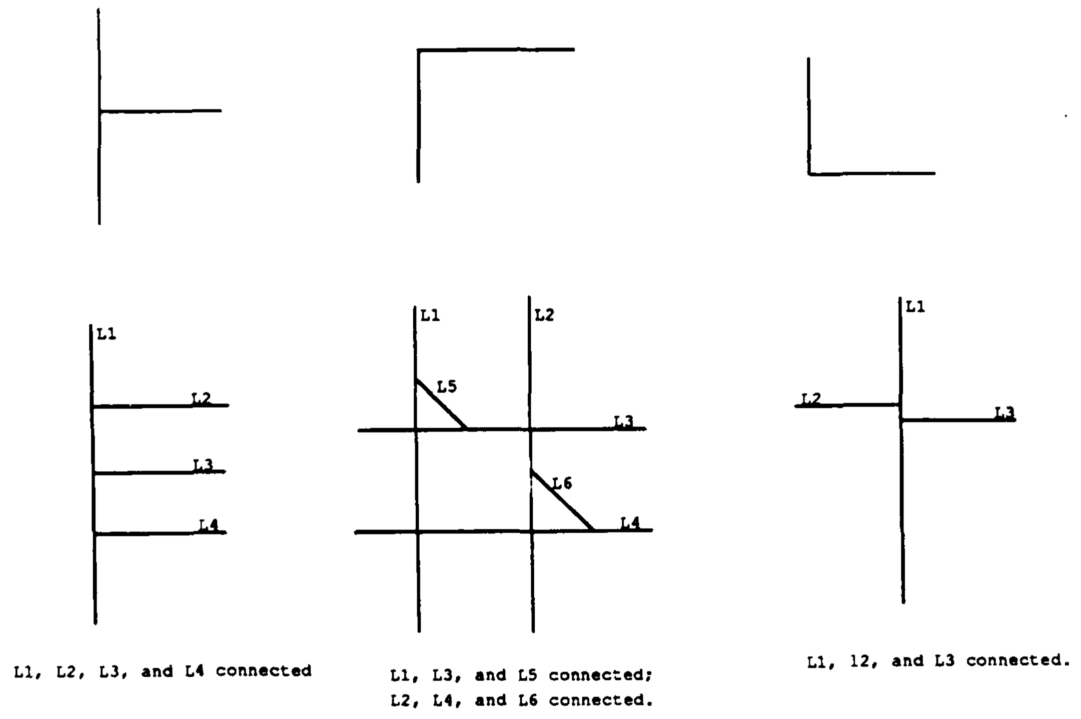
Wires carrying shadow signals must be named by:

```
/?
```

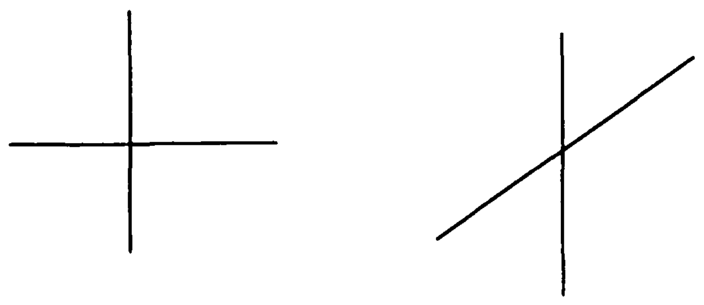
Whenever **dp2wl** sees such a string, it will create a unique name for the wire and declare the name as a shadow signal.

3.1.3 Unwired Pins

Another way of specifying an actual signal parameter is to directly place the name close to the pin. This can be strictly defined as:


L1
L2
L3
L4
L1, L2, L3, and L4 connected
L1
L2
L5
L3
L6
L4
L1, L3, and L5 connected;
L2, L4, and L6 connected.
L1
L2
L3
L1, 12, and L3 connected.

(a) touching Cases



(b) Non-touching Cases

Figure 4: *Examples of Line-Touch-Line*

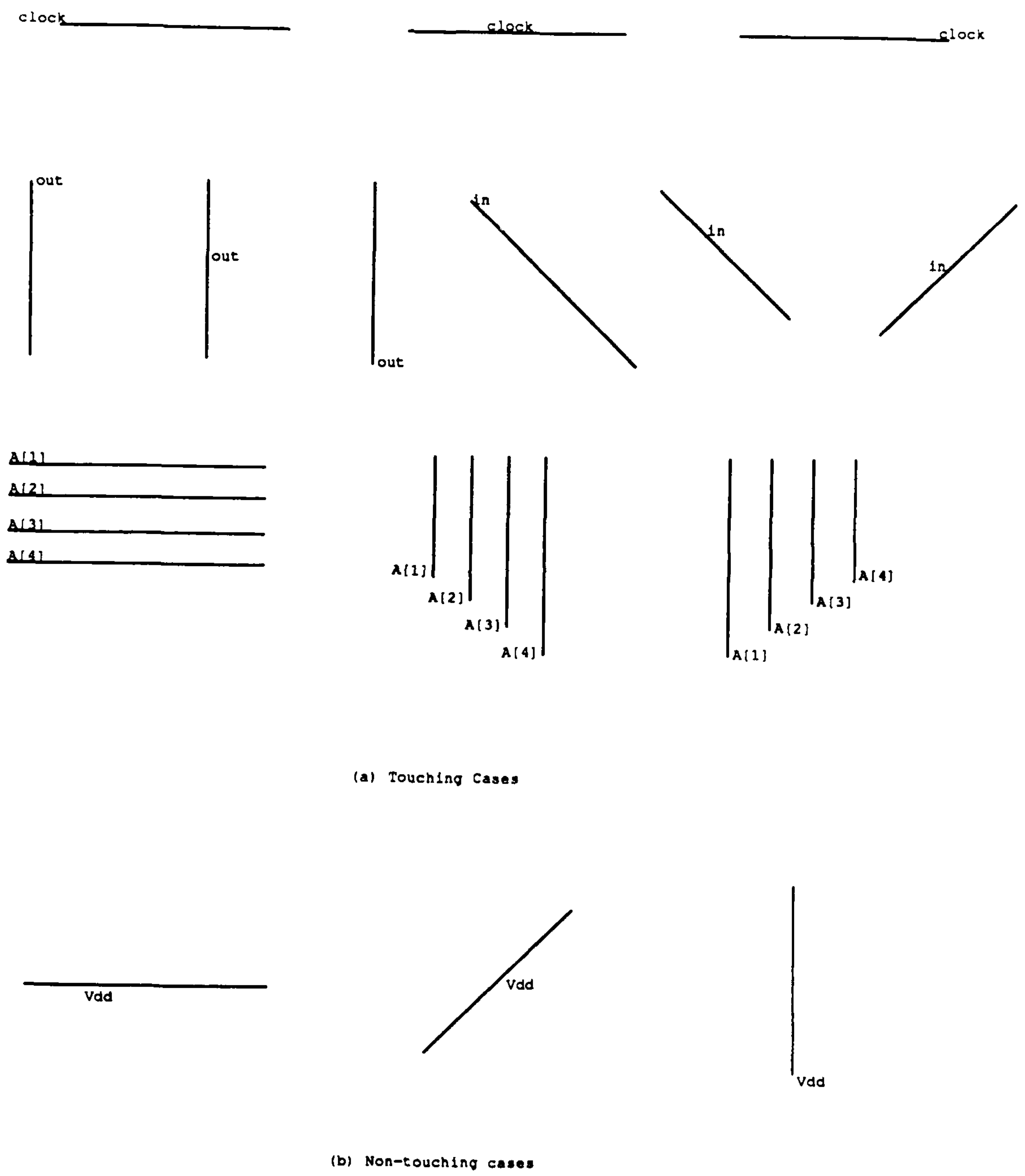


Figure 5: *Examples of String-Touch-Line*

Rule 6: string-touch-pin
Either the lower left corner or the lower right corner of the string should be within 3 pixels of the pin.

Note that the string here is *not* packed inside any symbol. If a string touches more than one pin, **dp2wl** will assign it to one pin and give a warning unless they are overlaid with each other (see below).

If two pins are sufficiently close to each other, they will be considered to be connected. This can be strictly defined as:

Rule 7: pin-touch-pin
One pin is within 3 pixels of another pin.

If there is no name associated with a group of connected pins, those pins are treated as if they were connected by unnamed wires. That is, **dp2wl** will create a unique name for them and declare the name as a short simple signal.

3.1.4 Default Names

If the name of an actual signal parameter is the same as that of the corresponding formal parameter, the user may simply leave the pin unconnected. **dp2wl** extracts the formal parameter name and uses it as the actual signal name. For example, in the drawing of **pipereg** (Figure 6), the actual signal parameters supplied to the device **pipe** are: `out[i], in[i], load, clk`.

3.2 Macro Blocks

Including Lisp inside drawings is trivial, but embedding drawings in Lisp requires a special mechanism called a *macro block*. Graphically, a macro block is a rectangle composed of width 2 lines which encloses a set of drawing items. It is named by placing a string of the form
{ *block-name* }
inside the rectangle area (see Rule 2). For example, in Figure 6, there is a macro block named **pipe**. Any reference to the block name, such as in
`(repeat i 1 size {pipe})`
will be substituted by the content of the block. This is a simple textual substitution. For example, if a wire name contains a variable, such as `in[i]` where `i` is the loop control variable, then different wires will be actually referenced in different iterations. However, if the wire name contains no variable, such as `load`, then the same wire will be referenced by each iteration. This is also the case for unnamed wires appearing in a macro block. Since this is a common source of errors, **dp2wl** requires the user to explicitly name all wires in macro blocks.

Nested blocks, as shown in Figure 7 (a), are allowed. However, the nesting in Figure 7 (b) is illegal.

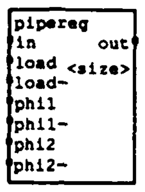
3.3 Signal Structure Declaration

Simple signal declarations can be easily specified in text, such as
`(local temp)`
in Figure 1. However, complicated signal structures, such as
`(access (sysBus (SW (ctrl (SW R/W strobe ack) (addr (BUS 0 23)) (data (BUS 1 32))))))`
are more easily defined graphically. Both **access** and **local/slocal** declarations are drawn similarly using a special type of symbol called *declaration symbols*. [2] We illustrate their use in **access** declarations, but these rules apply to **local/slocal** declarations as well by simply replacing the reserved word **access** by **local** or **slocal**.

[2] This feature is not ready for use at the moment.

pipereg

; pipeline register: drives only on phi2



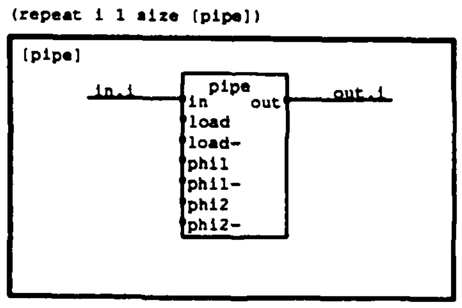
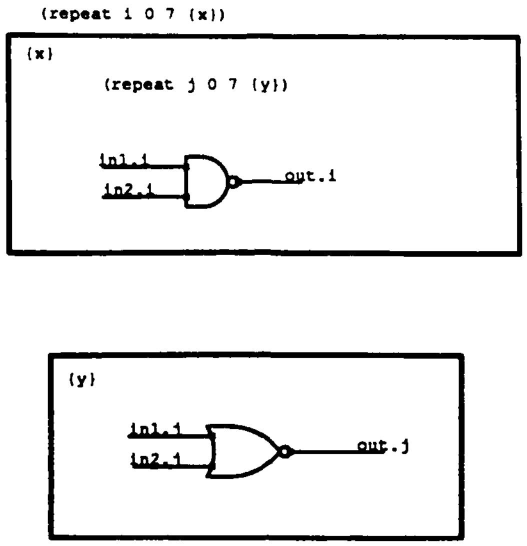
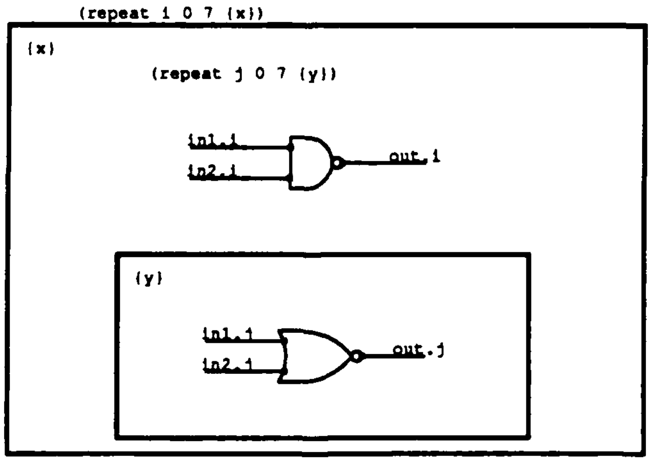


Figure 6: *A Wirelisp Drawing for* **pipereg**



(a) Nested Block References



(b) Illegal Nested Blocks

Figure 7: *Examples of Nested Blocks*

A declaration symbol designates one **access** declaration. It can be of any shape but must be named:

access:*type*. *

where, *type* is either BUS or SN, and anything after the period is merely a comment. Pin 0 represents the signal being declared, and other pins represent the fields of an SN structure or the lower and upper bounds of a BUS structure. The SN fields are ordered according to increasing pin numbers. For BUS structures, pin 1 always represents the lower bound, and pin 2 the upper bound. Figure 8 (a) gives a few examples.

There are two ways to associate a name with a pin in a declaration symbol:

1. Connecting the pin to a named wire (see Rule 4 & 5);
2. Associating the name with the pin directly (see Rule 6).

If multiple signals are declared simultaneously, then their names, separated by one or more spaces, must appear in a single **Xdp** string such as **x y** in Figure 8 (a).

Hierarchical structures can be drawn with the help of ***sub-structure symbols***. These symbols can be of any shape but must be named:

sub:*type*. *

Pin 0 represents the component whose sub-structure is being declared, and other pins represent the detailed structure just like in a declaration symbol. To specify a sub-structure for BUS type signals, the user must connect pin 0 of the sub-structure symbol to the lower bound pin (see Figure 8 (b)). Similarly for an SN type signal, pin 0 of a sub-structure symbol must be connected to the pin corresponding to the designated SN field. If multiple SN components share the same sub-structure, then their names, separated by one or more spaces, must appear in a single **Xdp** string.

The orientation of a declaration symbol or a sub-structure symbol does not matter.

3.4 General Functions

Arbitrary functions can be drawn in a manner similar to devices by using ***function symbols***.

A function symbol can be of any shape but must be named:

func:*func-name*. *

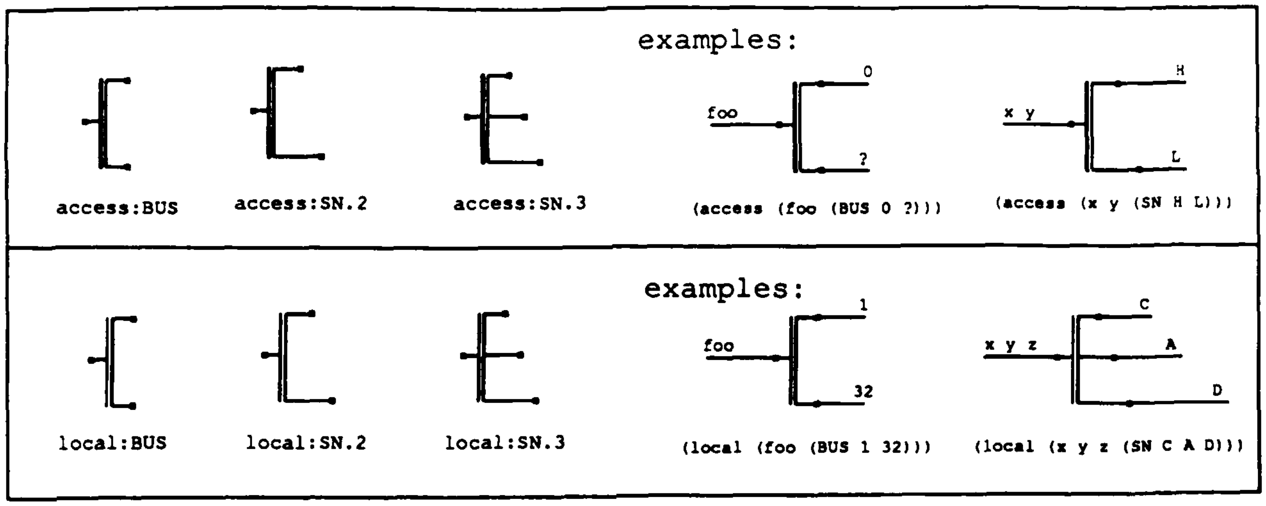
where, *func-name* names the function being called, and anything after the period is merely a comment. Pin 0 of the symbol represents the result of the function call, and other pins represent the parameters of the call. The order of parameters is determined by the increasing order of the corresponding pin numbers. A function may have no parameters but it must have exactly one output.

Names are associated with pins of function symbols just as in declarations. A function symbol will be translated into a function call:

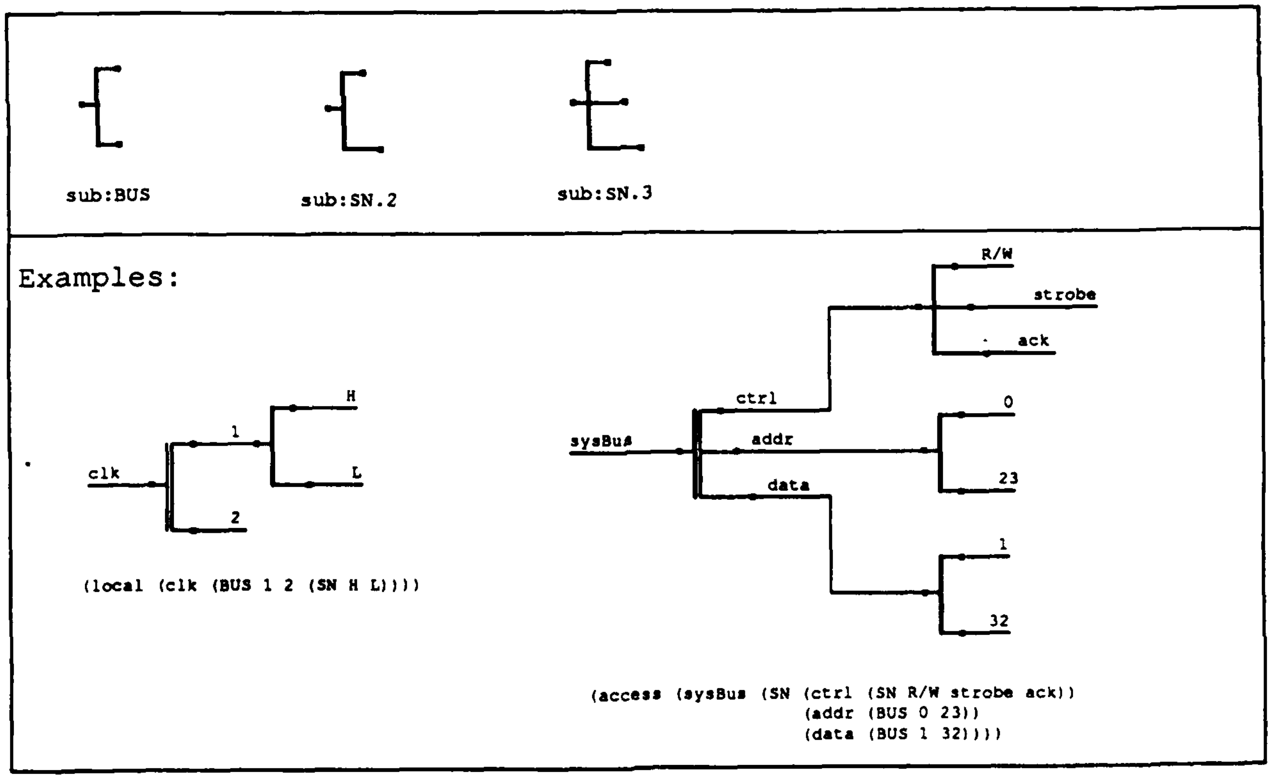
(*func-name input1 input2 input3* ...)

where, *input1, input2, input3* ... are pin names ordered according to increasing pin number. If pin 0 (output) does not have a name, this function call will be used to name the pin and any wire or pin connected to it. That is, if the pin is referenced twice, this function call will be evaluated twice which may produce different results. However, if pin 0 does have a name (which does *not* need to be declared anywhere), that name will be used as a temporary variable to cache the function return value. That is, the function will be evaluated only once before any device is instantiated but after all signal declarations have been evaluated. If pin 0 is not connected to anything, then the function symbol has no effect.

The most common use of function symbols is to represent signal operations. Figure 9 (a) gives several examples. The bundling operation (-B-), which is probably the most commonly used operation, can also be expressed in a simpler way: the user may name the wire with a string which is a list of space-separated signal names (Figure 9 (b)). Each signal name may be a complicated expression as long as it is evaluated to a signal. These signals are bundled together to form a new structured signal.

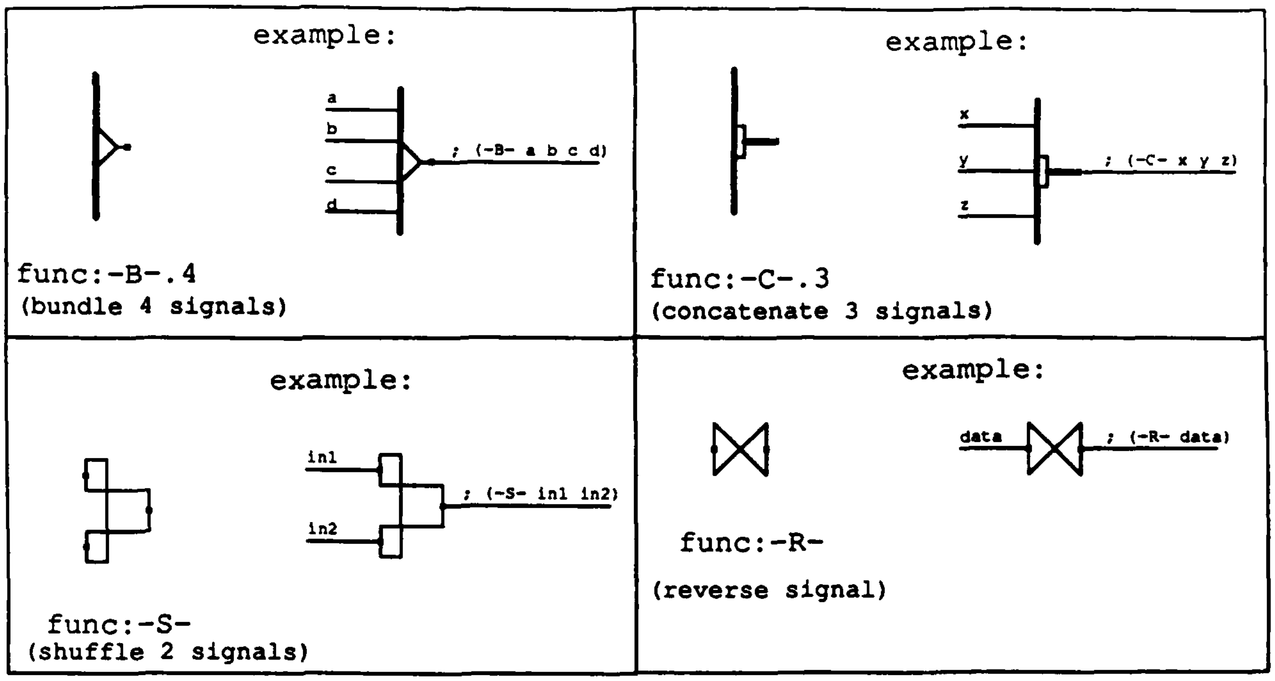


(a) simple access/local/slocal declarations

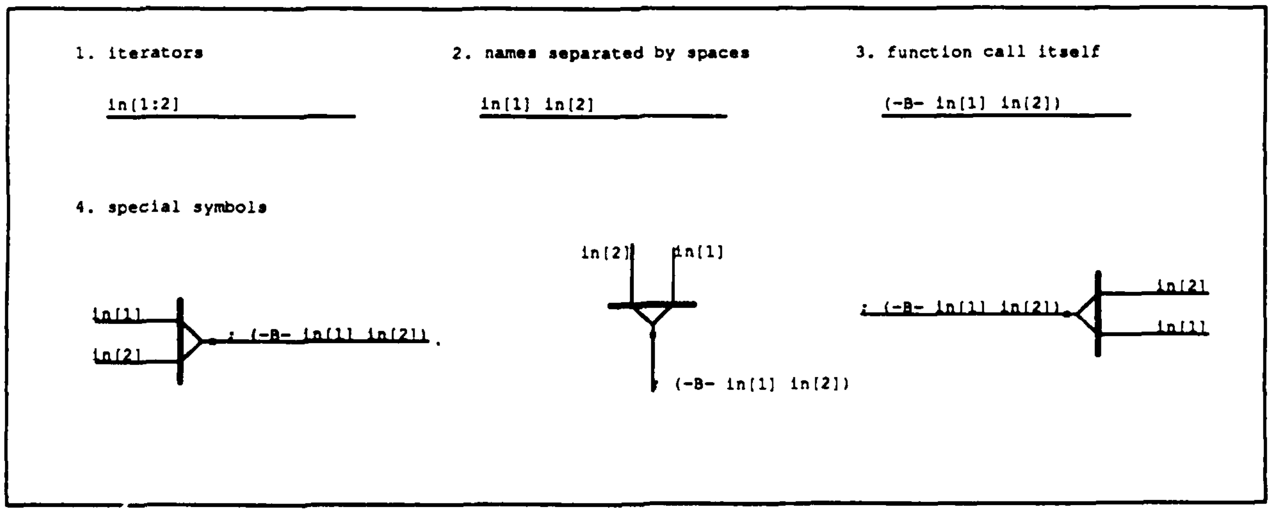


(b) hierarchical access/local/slocal declarations

Figure 8: *Drawing of Declaration Symbols*



(a) examples of function symbols



(b) various ways of bundling signals: (-B- in[1] in[2])

Figure 9: *Examples of Drawing Functions*

4 Special Cases: Foreign Devices and COSMOS Functions

A foreign device is defined in the same way as other devices except the drawing includes only a partial specification: the device interface and signal declarations.

Instantiating a foreign device is done by placing the special string

⟨`foreign`⟩

over the symbol (see Rule 2).

COSMOS functions can be drawn very much like a device instantiation except there must be a special string placed over the symbol (see Rule 2). This special string is in the form

⟨`cosmos:`n_1`:`n_2`:`n_3⟩

where, n_1, n_2, n_3 define the number of input signals, zero-delay signals and output signals respectively. That is, if pin names are ordered by the increasing order of pin numbers, then the first n_1 names will be viewed as input signals, the next n_2 names will be viewed as zero-delay signals, and the last n_3 names will be viewed as output signals. The instance name of the COSMOS function block is generated automatically by Wirelisp.

5 Output Files

After making a drawing, the user must save it in a file with the same name as the defined device. For example, the drawing in Figure 6 is saved in *pipereg.dp*. **dp2wl** translates a drawing into an equivalent Wirelisp procedure and saves it in the file with the same name but using the extension *.wl*, such as *pipereg.wl*. In the *.wl* file, **dp2wl** outputs the device header first, then all the declarations, both user-specified and automatically generated, in the order of:

all `access` declarations;
all `local` declarations;
all `slocal` declarations;
all `shadow` declarations;
all `defvar` declarations.

After that, all Lisp expressions are listed in decreasing Y-coordinate order. Macro block names appearing in these expressions are replaced by the contents of the corresponding blocks. Then all the device instantiations, including foreign devices and COSMOS functions, are listed in decreasing Y-coordinate order. Finally, all the aliasings (`-connect-`) are included. **Caution**: Device instantiations are evaluated in the order as listed. If shadow variables are used, then the user must make sure that the relevant devices are called in the right order.

If any Lisp expression is placed above the top symbol, it will appear before the device header.

6 Running dp2wl

Normally, **dp2wl** is invoked automatically by the Wirelisp interpreter. However, the user may run **dp2wl** independently by typing:

`dp2wl` *dpfile*

For example, to run **dp2wl** on *foo.dp*, the user may type:

`dp2wl foo`

which produces the output file *foo.wl*. If there is any error, an *.err.dp* file will be generated which pinpoints the trouble items.

dp2wl accepts the following flags:

- -d: create the *.wl* file if possible, even though errors exist.
- -g ⟨*a_string*⟩: append the given string to the device name and insert the expression
  `(include ''`*extended_device_name*`.ext'')`

at the end of the procedure. This is used exclusively for foreign devices.

7 Errors and Warnings

dp2wl generates error and warning messages when it finds problems with a drawing. Errors found by **dp2wl** should be corrected before continuing the execution. Currently **dp2wl** detects two types of errors:

1. A pin has no associated name.
2. A macro block contains an automatically generated signal with no indexing.

   Macro blocks that appear in repetitive statements are evaluated many times. If a wire has no name or is named by an integer, the same wire is used for all instances. Since this is usually not intended, the user must specify the name explicitly. Note that **dp2wl** cannot detect the error in `/#[i]` where `i` happens to be a constant.

dp2wl also gives following warnings:

1. A pin is connected to more than one wire.
2. A string is ambiguous.

   That is, it can be associated with more than one wire or pin. The user should make sure that the choice made by **dp2wl** is indeed what was wanted.
3. A general parameter string (enclosed by '(' and ')') is not associated with any device call.

   That is, the string is not sufficiently close to any symbol (see Rule 2). Note that if a name is intended to be a signal but not placed close enough to the wire or pin, it will be listed as a Lisp expression in the output because **dp2wl** cannot distinguish signals and Lisp expressions.
4. A pin appears at the top-level.

   Top-level items refer to those that are not packed inside any symbol, such as wires. However, pins *must* be packed inside some symbol. It is a common error that some pins are left floating around, since two pins on top of each other are not visible on the display even though they do exist.

When unnamed wires are frequently used, it is not unusual to forget to name a wire. To reduce this possibility, **dp2wl** reports how many automatic signal names are generated, including those for wires annotated by special strings like `/#` or `/?`. It also reports how many pins are named by default.

dp2wl relates the error and warning messages back to the drawing by writing an error **Xdp** file (with the extension *.err.dp*). This file stores the errors and warnings by placing them on different layers according to their type. The name of a layer tells the user what type of error or warning is on that layer. The errors and warnings can be viewed by first reading the original drawing and then overlaying it with the error file by the "*insert file*" entry of the command '**k**'. Then using the 'v' command, the user should turn off the *alter* capability for all layers except the layer of interest. Finally the 'S' command will highlight all errors of that type. This pinpoints the exact location of all errors.

All the errors and warnings (including unnamed wires) are reported graphically in the corresponding *.err.dp* file.

A Summary of Drawing Rules

Rule 1: string-touch-pin-inside-symbol

1. *A pin placed on the top (bottom) boundary of the symbol will be associated with the closest string below (above) whose left and right ends span the pin's X location.*
2. *Otherwise, a pin placed to the left (right) of the symbol center will be associated with the closest string to its right (left) whose top and bottom corners span the pin's Y location.*
3. *In both cases, the string must be within 12 pixels of the pin.*

Rule 2: string-touch-box
Either the lower left corner or the lower right corner of the string should be inside the area enclosed by the box.

Rule 3: line-touch-line
Either end of one line should be within 3 pixels of some point on the other line.

Rule 4: line-touch-pin
The pin should be within 3 pixels of some point on the line.

Rule 5: string-touch-line
Either the lower left corner or the lower right corner of the string should be within 3 pixels of some point on the line.

Rule 6: string-touch-pin
Either the lower left corner or the lower right corner of the string should be within 3 pixels of the pin.

Rule 7: pin-touch-pin
One pin is within 3 pixels of another pin.

B New Xdp Commands

At the University of Washington, **Xdp** has been mostly used to make hierarchical circuit drawings, where a set of devices form a tree hierarchy. It is frequently the case that the designer needs to traverse the hierarchy to make various changes. Several new **Xdp** commands have been added to make this easier, as shown below.

B.1 Sub-edit Symbols: 'E' , 'q' and 'Q'

These commands help the user traverse the device hierarchy to make changes without manually saving or reading the drawings.

Press the key 'E' to enter *sub-edit symbol* mode. To edit a device, move the cursor over the symbol representing that device and press any mouse button. **Xdp** will erase the screen and read the drawing of that device. It is assumed that the drawing of a device is saved in a *.dp* file with the same name as the device. This file should be placed either in the current working directory or in the directories listed in **DPPATH**. If the current window has been modified since last saved, **Xdp** will automatically save the drawing in the corresponding *.dp.CKP* file. If the sub-edited drawing file is found, the user may continue as if just entering **Xdp**. Otherwise, **Xdp** will prompt the user for another file name.

To exit from one level of sub-editing, the user uses the command 'q'. **Xdp** will erase the screen and read the drawing for the device on the top of sub-editing stack. If this is already the root device, **Xdp** exits.

Also, **Xdp** will ask the user whether or not to redefine the symbol just sub-edited. Answer *yes* if the symbol definition was modified. The default is *no* to avoid files being needlessly modified.

To exit all levels of sub-editing except the root device, the user may use the command 'Q'. It clears the sub-editing stack and restores the drawing for the root device.

The user may also view which devices are on the stack by the command 'k' ('*show subedit stack*' entry).

B.2 Redefine Symbols: 'y' and 'Y'

'y' is used to redefine one symbol at a time, while 'Y' performs a recursive redefinition.

Press the key 'y' to enter *redefine symbol* mode. Move the cursor over the (device) symbol to be redefined and press any mouse button. **Xdp** will read in a new definition from the corresponding *.dp* file. If the file is not found, **Xdp** will prompt the user for another file name and try again.

When the key 'Y' is pressed, every device in the symbol hierarchy is redefined. The search is depth-first and each device is redefined exactly once. Before updating each device, **Xdp** asks the user whether or not it should be changed. If the drawing file is not found, **Xdp** simply skips to the next device.

B.3 Other Commands: 'I' and 'C'

Command 'I' has been changed to *erase the screen and read the given file*. The original file inserting function has become an entry (*insert file*) of the special command 'k'.

Command 'C' can be used to clear the screen.

ACKNOWLEDGEMENT

We are very grateful to Larry McMurchie and Robert Condon for their help in preparing this manual and experimenting **dp2wl**.